Degeneracy in Geometric Computation and the Perturbation Approach

PETER SCHORN
Institut für Theoretische Informatik, ETH, CH-8092 Zürich, Switzerland

We study the problem of degeneracy in geometric algorithms and show that degeneracies arise even in simple Euclidean constructions with ruler and compass. We distinguish between problem-dependent and algorithm-dependent degeneracies, and argue that the popular perturbation approach is suitable for removing only the latter but not the former. Examples demonstrate the dangers of removing problem-dependent degeneracies using the perturbation approach and we identify circumstances where this method is justified. We propose to deal with degeneracies by giving precise input–output specifications of the geometric problem under consideration and by handling problem-dependent degenerate cases individually right from the beginning of algorithm construction. Algorithm-dependent degeneracies are removed using perturbation or its simplest version, lexicographic ordering. As an example of this approach we present an algorithm for the computation of the winding number which yields provably correct results when implemented in integer and floating point arithmetic.

1. THE PROBLEM OF DEGENERACY IN GEOMETRIC COMPUTATION

Implementors of geometric algorithms will testify that special cases, or degeneracies, often cause the greatest difficulties when creating a program that works for all inputs. In order to illustrate the concept of degeneracy consider the following Euclidean construction which can be viewed as a tiny geometric algorithm. Given a point p and a circle c with positive radius, find a line through p that touches c (i.e. a tangent). The standard construction, as depicted in Figure 1, works as follows:

1. Draw the straight line segment l from the center of c to p.
2. Construct m, the middle point on l, by intersecting the circle around p with radius $r = \mathrm{d}(p, \mathrm{center}(c))$ and the circle around center(c) with radius r and connecting the intersecting points with a straight line segment k. The intersection of l and k is m.
3. The touching points of the tangents on c are given as the intersection points of c with the circle around m with radius $\mathrm{d}(p, m) = \mathrm{d}(\mathrm{center}(c), m)$ (d denotes the Euclidean distance).

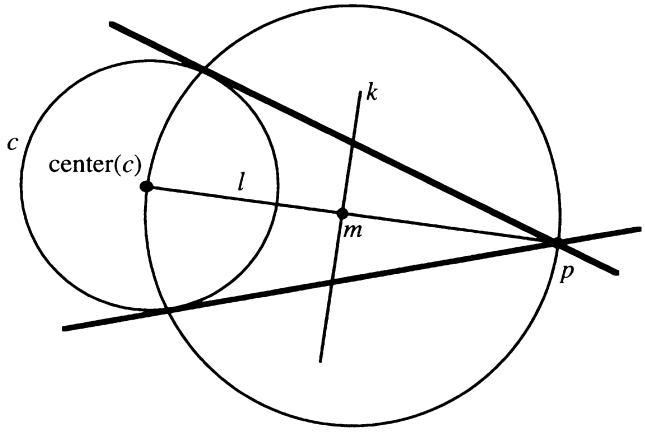

FIGURE 1. Constructing tangents with compass and ruler.

Note that this construction works well when p is outside the circle (two tangents exist), gives the right answer when p is inside the circle (no tangent exists) but fails to handle the degenerate case where p lies on c. In this special case, the intersection of c with the circle around m with radius $\mathrm{d}(p, m)$ yields p itself (see Figure 2) which is insufficient to determine a tangent. In this case we need to find the perpendicular to l through p, a task easily accomplished by ruler and compass (Figure 2).

This example captures two essential characteristics of geometric degeneracy:

1. Degenerate cases occur even for the simplest geometric algorithms. In fact, geometric configurations arising from real world applications tend to favour degenerate configurations for two reasons:
   a. The underlying number system (floating point or integer) severely restricts the locations of geometric objects to a finite grid. As a consequence, the number of collinear points increases when point data from a larger universe is rounded to lie on the grid.
   b. Real world spatial data such as floor plans, blue prints, layouts, etc., exhibit a great deal of structure such as vertices incident several edges, parallel lines, etc. which is not found in random configurations.
2. There appears to be no straightforward way to modify the algorithm for the non-degenerate case in such a way that it handles the degenerate case as well. Therefore, special code may be required to deal with degenerate configurations. In our example, this is no accident since the case 'p lies on c' is fundamentally different from the cases 'p outside c' and 'p inside c'. It is the only case where a unique tangent exists.

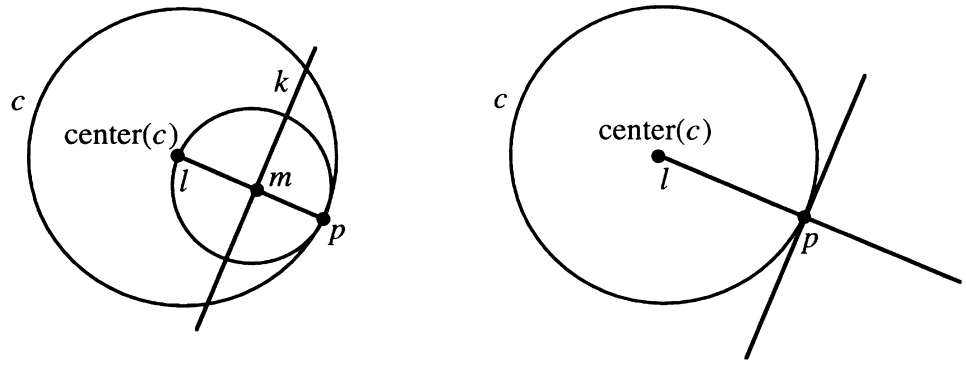

FIGURE 2. The degenerate case 'p lies on c' calls for a different procedure.

One of the recent developments in computational geometry was the introduction of (symbolic) perturbation for the removal of degeneracies. Great hopes in its capabilities can be found in Yap (1990, p. 351) where the perturbation approach is advertised as a contribution towards the goal 'to admit implementors to the theoreticians' paradise in which degeneracies are (virtually) abolished'. This paper takes a more cautious view. We distinguish between problem-dependent degeneracies and algorithm-dependent degeneracies, and claim that the latter can be removed by perturbation while the former are dealt with precise input–output specifications.

The paper is organized as follows. Section 2 introduces the idea of perturbation. Section 3 defines the notions of problem-dependent and algorithm-dependent degeneracies. Section 4 discusses the merits and drawbacks of the perturbation approach in geometric computation. As a case study, section 5 presents in detail an algorithm for the winding number and its robust implementation in double precision integer arithmetic and floating point arithmetic with guaranteed accuracy. Section 6 proposes how degeneracies should be dealt with in general.

2. THE PRINCIPLE OF PERTURBATION AND NUMERICAL ANALYSIS

The basic idea of the perturbation approach can be formulated as follows:

Principle of perturbation The approximate result computed by a program can be viewed as the exact result on slightly perturbed input data.

In numerical analysis this method is known as backwards analysis and enables the analysis of roundoff errors in floating point arithmetic. A simple example is the fundamental property of floating point addition. Let $\oplus$ denote floating point addition and *eps* the machine epsilon ($eps = b^{1-p/2}$, where b is the base and p the number of digits in the mantissa). Then the equation

$$a \oplus b = (a+b)\cdot(1+\varepsilon) = a\cdot(1+\varepsilon) + b\cdot(1+\varepsilon)$$
$$\text{with } |\varepsilon| \leqslant eps$$

holds.

We interpret this formula such that the result of a floating point addition of a and b is the exact sum of two slightly perturbed numbers $a\cdot(1+\varepsilon)$ and $b\cdot(1+\varepsilon)$. Numerical analysis favors algorithms which can be analyzed using backwards analysis and tries to bound the amount of perturbation necessary to achieve exact results.

The well known idea of lexicographic ordering can also be viewed as perturbation in disguise. Consider a plane sweep algorithm where the sweep line moves from left to right, processing events according to their x-order. This works well until two or more events with the same x-coordinate exist (see Figure 3). A common remedy is to break ties by taking the y-order into account (i.e. lexicographic ordering) which can be viewed as a perturbation (see Figure 3).

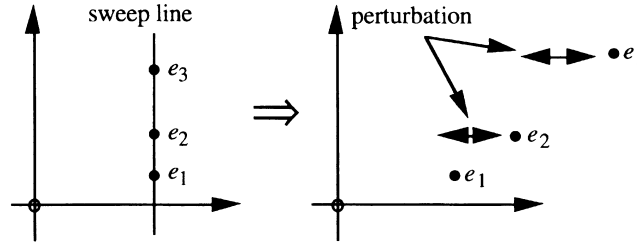

FIGURE 3. Lexicographic ordering viewed as perturbation.

Before we can apply perturbation to geometric algorithms in general we must analyze the notion of degeneracy.

3. PROBLEM-DEPENDENT AND ALGORITHM-DEPENDENT DEGENERACIES

Degeneracy of a configuration can be defined in at least two different ways (Yap, 1990).

The *first definition* is geometry centered. We call configuration C degenerate with respect to a problem P iff the solution of P is sensitive to some infinitesimal perturbation of C, i.e. if for any $\varepsilon > 0$ there exists a perturbation bounded by ε leading to a topologically different result. For example, a configuration falling into T_2 in Figure 4 is degenerate because the change of p_4's x-coordinate by any amount leads to a new classification. In other words, a degeneracy is a discontinuity in the input–output mapping.

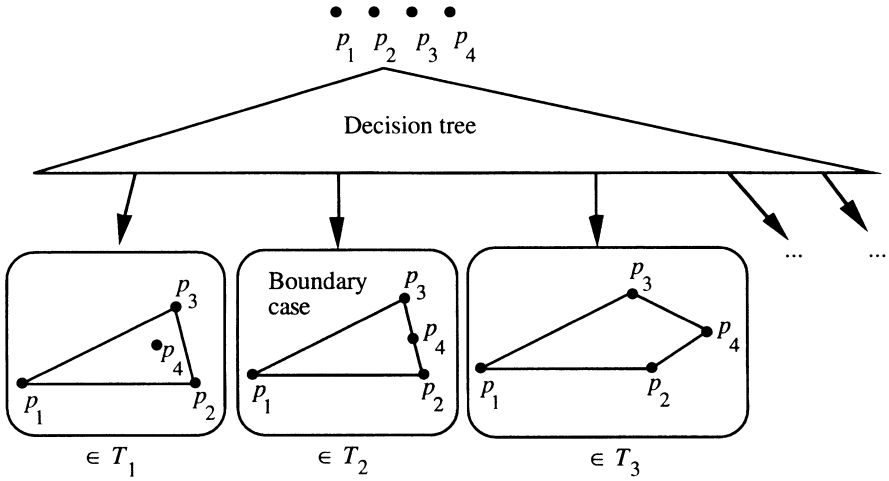

FIGURE 4. Partial decision tree for the convex hull problem.

The *second definition* is algorithm centered. We use the algebraic decision tree model of computation which views all possible executions of a program as a tree where algebraic functions govern which branch to take. We think of a geometric algorithm A in terms of this model. Consider the inner nodes of the decision tree where some of the nodes contain predicates on the input data. For the sake of simplicity we assume these predicates to be testing the sign of a polynomial in many variables. For example, the sign of the polynomial $(p_x - q_x)(q_y - r_y) - (p_y - q_y)(q_x - r_x)$ determines on which side of the directed line from point q to point r the point p lies. A configuration C is called degenerate with respect to algorithm A iff at least one of the test polynomials encountered during A's execution evaluates to zero. For example, the polynomial from above evaluates to zero iff the points p, q and r are collinear. A typical example of an algorithm-dependent degeneracy for a plane sweep algorithm is a configuration of points where more than two points share the same x-coordinate.

A degeneracy according to the first definition is called a problem-dependent degeneracy while a degeneracy according to the second definition which is not problem-dependent is called algorithm-dependent. One can see that degeneracies according to the first definition are always degeneracies according to the second definition: a degenerate configuration according to the first definition (e.g. to be found in T_2 in Figure 4) can be viewed as a boundary case separating two types T_1 and T_3 of solutions with distinct topological properties. Since an algorithm A claiming to solve problem P must be able to distinguish between the two topologically distinct solutions, we can continuously change a configuration with result in T_1 into a configuration with result in T_3 forcing an appropriate test predicate to evaluate to zero for a configuration in T_2.

A typical example of a problem-dependent degeneracy in the convex hull problem is a set of points where more than two points lie on a hull edge. From a user's point of view problem-dependent degeneracies are the main concern while algorithm-dependent degeneracies are effects of a particular problem solving approach which should be hidden from the user. Note also that non-degeneracy conditions imposed are often more stringent than our second definition: when constructing the convex hull for example, one might want to exclude all configurations of points which contain more than two collinear points, even though most of these configurations will neither be problem-dependent nor algorithm-dependent degeneracies.

4. PERTURBATION IN GEOMETRIC COMPUTATION

Many researchers in geometric computation have adopted the principle of perturbation as a sufficient requirement for a geometric algorithm to be robust and useful. This general principle promises to handle simultaneously three separate problems encountered when creating robust geometric programs:

1. The removal of algorithm-dependent degeneracies (e.g. vertical line segments in certain plain sweep algorithms).
2. The removal of problem-solving degeneracies (e.g. collinear points in a convex hull algorithm).
3. The ability to analyze programs built on imprecise primitives, implemented in floating point arithmetic.

Furthermore, general methods have been devised that solve these problems once and for all by building geometric algorithms on top of carefully designed primitives. For example, Edelsbrunner and Mücke (1990), Yap (1990) and Emiris and Canny (1992) address points 1 and 2, whereas Salesin *et al.* (1989) address 1, 2 and even 3.

4.1. The inadequacy of perturbation

In the following we argue that the principle of perturbation as stated above is inadequate for many geometric algorithms because we loose the distinction between

algorithm-dependent degeneracies and problem-dependent degeneracies. Therefore the general perturbation methods will indiscriminately remove both types of degeneracy. One might view this effect as a good thing, but in practice it is not. The treatment of problem-dependent degeneracies is not under the user's control and problem-dependent degeneracies which look very similar can be handled quite differently by an algorithm just adhering to the principle of perturbation. The following examples show what can happen if a geometric algorithm satisfies just the principle of perturbation. Similar views were expressed by Mehlhorn (1993).

Point-in-polygon test. Given a polygon P and a point p determine whether p lies inside P. If exact arithmetic is used (e.g. Edelsbrunner and Mücke, 1990) and the point lies on the boundary of the given polygon, the algorithm can randomly choose between 'in polygon' and 'outside polygon' without violating the principle of perturbation (see Figure 5). Although the schemes in Edelsbrunner and Mücke (1990), Yap (1990) and Emiris and Canny (1992) are deterministic, it is unclear for which points on the boundary the test yields 'inside' and for which 'outside'. A better solution to this problem would be to introduce a third possibility 'on boundary'. Edelsbrunner and Mücke (1990) acknowledge the existence of this problem and suggests testing 'on boundary' condition before running the point-in-polygon test. In Section 5 we give a more efficient algorithm, nevertheless based on the perturbation idea, that handles both the problem-dependent degeneracy 'on boundary' and any algorithm-dependent degeneracies.

The difficulties with the principle of perturbation increase when imprecise primitives are allowed as is the case in Salesin (1989). Here, a point close to the boundary of the polygon might be considered 'inside' or 'outside' without giving the user an adequate warning that a nearly degenerate configuration has been encountered (see Section 5 for a solution in floating point arithmetic that either gives the correct result or detects near degeneracy).

Convex hull. Find the smallest convex polyhedron C that contains a set of points in d dimensions. Again, we encounter the situation that problem-dependent degeneracies, i.e. in this case points that lie on C without being vertices of C, are treated arbitrarily. A possible better solution is to return C without any extraneous points and to return other points on C separately. Unfortunately it seems rather complicated to implement this specification without taking this problem-dependent degeneracy explicitly into special account. Note, in contrast, that the perturbation approach works perfectly well for an algorithm that computes just the volume of C because points on C are not problem-dependent degeneracies but might be viewed as algorithm-dependent degeneracies.

Line segment intersection. Compute all intersections of n given line segments in the plane. Using the perturbation approach we might encounter difficulties with line segments that overlap or merely touch each other. In both cases an algorithm satisfying the principle of perturbation is free to decide whether the segments intersect or not. In this case post-processing, i.e. detecting degeneracies, appears as hard as the original problem and is therefore infeasible. Again, a more precise specification of the desired output in problem degenerate cases helps to solve the problem: one can define that two line segments intersect iff there exists at least one point common to both segments.

4.2. The adequacy of perturbation

The preceding examples show that indiscriminate use of the perturbation approach can lead to difficulties. This section identifies two circumstances under which the perturbation approach is valid.

1. A precise specification of the result in case of problem-dependent degeneracies may be unnecessary — any solution is good enough. As an example, consider computing the volume of the convex hull C of a given point set in three dimensions. As an intermediate step we employ a convex hull algorithm. Extraneous points on C pose no problem since, in the worst case, they could cause zero-volume tetrahedra if the volume is computed using a tetrahedralization of C. Other examples include extremal problems, like closest-pair, Euclidean-minimum-spanning tree, etc., where the solution is not necessarily unique.
2. Although perturbation is used, the problem-dependent degeneracy can easily be recognized in the output. An example is the computation of the Voronoi diagram of a set of four co-circular points (see Figure 6). Using perturbation we would obtain a Voronoi diagram with a zero-length Voronoi edge. In a post-processing stage we could fuse the two Voronoi

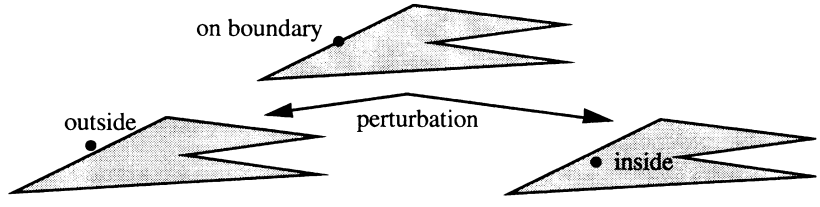

FIGURE 5. Perturbation for the point-in-polygon test.

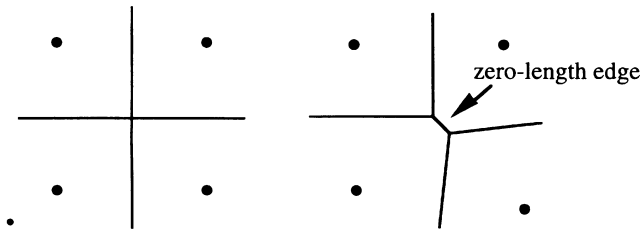

FIGURE 6. Perturbing a degenerate Voronoi diagram.

vertices or even allow zero-length Voronoi edges facilitating the computation of the Delaunay triangulation. Note that in the examples where we encountered difficulties with the perturbation approach, the computed output did not reliably indicate problem-dependent degeneracies. For example, in the line segment intersection problem we can miss some degenerate intersections and there is no way to detect this fact afterwards.

5. CASE STUDY: COMPUTING THE WINDING NUMBER ROBUSTLY

The winding number is a generalization of the point-in-polygon test, a function often found in computer graphics. This case study demonstrates how to compute the winding number such that problem- and algorithm-dependent degenerate cases are handled appropriately. We give an exact version implemented in integer arithmetic and a floating point arithmetic version with guaranteed accuracy. We use the perturbation approach for algorithm development.

Definition (*winding number*). Given a polygon P, not necessarily simple, and a point t not on P's perimeter, the winding number $\mathrm{wn}(P, t)$ is the number of complete revolutions in the positive direction an observer in t has to make in order to follow an object that travels on P's perimeter in the positive direction until it returns to its starting point. ■

5.1. The ray shooting method

The standard algorithm for computing the winding number works as follows. We choose a directed half ray r emanating from t and initialize a variable wn to 0. For each directed edge of P that crosses r from left to right, looking in direction r, we increment wn, and for each directed edge that crosses from right to left we decrement wn. Edges not intersecting r are ignored. After examining all edges of P the variable wn contains the winding number. The choice of the ray r is arbitrary, a horizontal or vertical ray is efficient.

Instead of writing special code for the special cases (e.g. the ray touches a vertex), we use a symbolic perturbation approach (Edelsbrunner and Mücke, 1990). We take a horizontal half ray emanating at $t = (t_x, t_y)$ and going to $(t_x, t_y + \varepsilon)$ and from there to $(+\infty, t_y + \varepsilon)$ where ε is a symbolic quantity only affecting the borderline cases of comparisons (see Figure 7). In order to detect whether a horizontal half ray intersects an arbitrary line segment, the primitive $\mathrm{whichSide}(p, q, r) = (p_x - q_x)\cdot(q_y - r_y) - (p_y - q_y)\cdot(q_x - r_x)$ *suffices*. The polynomial $\mathrm{whichSide}(p, q, r)$ is positive if q lies to the right of the directed line from p to r, is negative if q lies to the left of this line and is zero if q lies on the line. In contrast to Edelsbrunner and Mücke (1990) which employs a separate pre-processing phase to detect the case where t lies on P's boundary, we merge the two phases thereby enhancing the algorithm's efficiency. We must distinguish three cases (Figures 8–10).

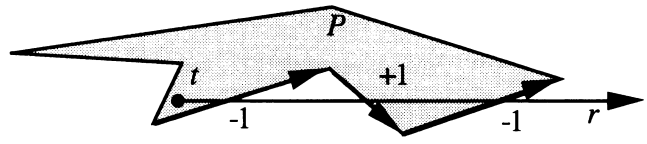

FIGURE 7. The half ray method for computing the winding number.

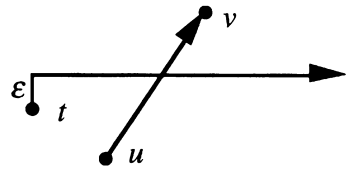

FIGURE 8. The edge (u, v) crosses from right to left.

$$w_n := w_n + 1 \Leftrightarrow (\mathrm{whichSide}(u, t, v) < 0) \wedge (u_y \leqslant t_y < v_y)$$

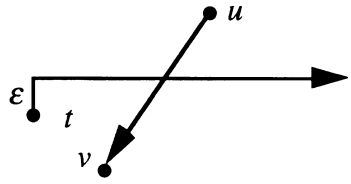

FIGURE 9. The edge (u, v) crosses from left to right.

$$wn := wn - 1 \Leftrightarrow (\mathrm{whichSide}(u, t, v) > 0) \wedge (v_y \leqslant t_y < u_y)$$

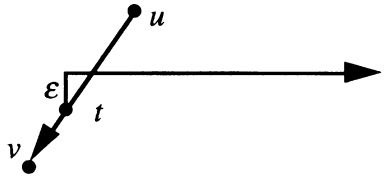

FIGURE 10. The ray emanates on an edge.

$$onBoundary \Leftrightarrow (\mathrm{whichSide}(u, t, v) = 0) \wedge (((u_y = v_y) \wedge ((u_x \leqslant t_x \leqslant v_x) \vee (v_x \leqslant t_x \leqslant u_x))) \vee ((u_y \neq v_y) \wedge ((u_y \leqslant t_y \leqslant v_y) \vee (v_y \leqslant t_y \leqslant u_y))))$$

5.2. The integer arithmetic case

Tying the above three cases together, we obtain the following program.

```
procedure windingNumber(
  P       : array of point; { Vertices of the polygon                          }
  n       : integer;        { Number of vertices                               }
                            { Precondition: (∀ i: 1 ≤ i < n: P_i ≠ P_{i+1}) ∧ (P_1 ≠ P_n)}
  t       : point;          { We seek the winding number around t.             }
  var on  : boolean;        { 'true' ⇔ t is on P's boundary                    }
  var wn  : integer);       { The winding number                               }
begin
  wn:=0; u:=P[1]; P[n + 1]:=u; {Sentinel} on:=false;
  for i:=2 to n + 1 do
    v:=P[i];
    case sign(whichSide(u, t, v)) of
    −1:   if u_y ... t_y < v_y then wn:=wn + 1 end
{D} |0:   on:=on  ∨  ((u_y = v_y) ∧ ((u_x ≤ t_x ≤ v_x) ∨ (v_x ≤ t_x ≤ u_x)))
                  ∨  ((u_y ≠ v_y) ∧ ((u_y ≤ t_y ≤ v_y) ∨ (v_y ≤ t_y ≤ u_y)))
    |+1:  if v_y ... t_y < u_y then wn:=wn − 1 end
    end;
    u:=v
  end
end;
```

Examining the formula $\text{whichSide}(p, q, r) = (p_x - q_x)(q_y - r_y) - (p_y - q_y)(q_x - r_x)$, we see that we can compute an exact result using double precision integer arithmetic if the vertices of polygon P have single precision integer coordinates.

5.3. An accurate floating point implementation

Now we consider the case of floating point arithmetic. The idea is to implement 'whichSide' in floating point arithmetic such that the design is either correct, i.e. as obtained with infinite precision arithmetic, or the computed value is zero and there exists a small perturbation that causes the three points to be collinear. We call the latter case a nearly degenerate case and find in the following analysis that nearly degenerate cases can be treated as exact degenerate cases. This works well in this example because this treatment does not really change the course of the algorithm. In fact one could deliver the result 'on boundary' as soon as a near or exact degeneracy has been encountered.

For the implementation we use floating point arithmetic which either rounds upwards or downwards in order to simulate some kind of interval arithmetic. The following error analysis uses the fact that

$$(a \text{ op}\downarrow b) = (a \text{ op } b)(1 - \varepsilon), \qquad 0 < \varepsilon < 2\,eps$$

(downward-rounding arithmetic) and

$$(a \text{ op}\uparrow b) = (a \text{ op } b)(1 + \varepsilon), \qquad 0 < \varepsilon < 2\,eps$$

(upward-rounding arithmetic)

with $\text{op} \in \{+, -, *\}$ where *eps* is the machine epsilon. An implementation of an arithmetic expression E in downward-rounding floating point arithmetic is denoted by 'downRound(E)' and an implementation in upward rounding floating point arithmetic is denoted by 'upRound(E)'. The following theorem describes a robust floating point implementation for the winding number problem and gives a bound on the perturbation necessary in degenerate cases.

Note that the algorithm, unlike algorithms built on ε-predicates in Epsilon geometry (Salesin *et al.*, 1989), explicitly detects the nearly degenerate cases where the answer of the algorithm is only correct if a small perturbation is applied. In cases where no near degenerate case is encountered, we obtain the same result as if we had used infinite precision arithmetic:

THEOREM (*accurate floating point implementation of the winding number*). Let 'windingNumberFloat' be the procedure obtained from 'windingNumber' by replacing the line

case sign(whichSide(u, t, v)) **of**

by

case SignWhichSideFloat(u, t, v) **of**

where

```
function SignWhichSideFloat(u, t, v: point): {−1, 0, +1};
begin
  swaps:=0;
  if |t_y| > u_y| then (u, t):=(t, u); swaps:=swaps + 1 end;
  if |t_y| > v_y| then (v, t):=(t, v); swaps:=swaps + 1 end;
  {|t_y| ≤ min(|u_y|, |v_y|), use for the error analysis.}
  if    downRound((u_x − t_x)*(t_y − v_y))
        > upRound((u_y − t_y)*(t_x − v_x))
  then  SignWhichSideFloat:=(−1)^swaps
  elsif upRound((u_x − t_x)*(t_y − v_y))
        < downRound((u_y − t_y)*(t_x − v_x))
  then  SignWhichSideFloat:=(−1)^(swaps+1)
  else  SignWhichSideFloat:=0
```

```
  end
end;
```

If after execution of the procedure 'winding NumberFloat' the variable *on* is 'false' then *wn* is the correct winding number of the polygon P around the point t, where the floating point numbers representing the coordinates of P's vertices and t are assumed to be exact. If *on* is 'true' then there is a perturbation bounded by 12 *eps* for two vertices v_1 and v_2 of P and the point t such that the perturbed point t^* lies on the perturbed edge e^* determined by v_1^* and v_2^*. ■

Proof. From the previous discussion and the construction of 'SignWhichSideFloat' it is clear that the results of -1 or $+1$ are correct in the sense that these variables would have been obtained using even infinite precision arithmetic. For the analysis we consider three cases.

1. The statement {D} is never executed. Then *on* = 'false' and the claim is correct according to the previous discussion.
2. The statement {D} is executed, but *on* = 'false' after completion of the program. In this case it is conceivable that *wn* should have been incremented or decremented. In the case $u_y = v_y$ the variable *wn* should not change because neither $u_y < v_y$ nor $v_y < u_y$ is satisfied. In the case $u_y \neq v_y$ we conclude that $\neg(u_y \leqslant t_y \leqslant v_y) \wedge \neg(v_y \leqslant t_y \leqslant u_y)$ (*on* remains 'false') which in turn prohibits changing the variable *wn*.
3. The statement {D} is executed and *on* = ≅ true. We consider the execution that changed the value of *on* from 'false' to 'true'. We can assume that $|t_y| \ldots \min(|u_y|, |v_y|)$ according to the definition of 'SignWhichSideFloat'. We use the following abbreviations:

$$a = (u_x - t_x)(t_y - v_y)$$

$$a\uparrow = \text{upRound}(a), \qquad a\downarrow = \text{downRound}(a)$$

$$b = (u_y - t_y)(t_x - v_x),$$

$$b\uparrow = \text{upRound}(b), \qquad b\downarrow = \text{downRound}(b)$$

From 'SignWhichSideFloat' we obtain $a\downarrow \leqslant b\uparrow \wedge b\downarrow \leqslant a\uparrow$. If we assume $a\downarrow \leqslant b\downarrow$ we get $a\downarrow b\downarrow \leqslant a\uparrow$. Otherwise we have $b\downarrow \leqslant a\downarrow$, resulting in $b\downarrow \leqslant a\downarrow \leqslant b\uparrow$. Since the cases $a\downarrow \leqslant b\downarrow$ and $b\downarrow \leqslant a\downarrow \leqslant b\uparrow$ are symmetric, we assume $a\downarrow \leqslant b\downarrow \leqslant a\uparrow$. Numerical analysis, omitting second order error terms and using the facts about upwards and downwards rounding floating point arithmetic shows that $a\downarrow = a(1-\varepsilon_1)$, $a\uparrow = a(1+\varepsilon_2)$ and $b\downarrow = b(1+\varepsilon_3)$ with $0 \leqslant |\varepsilon_i| < 6\,eps$. Since $a\downarrow \leqslant b\downarrow \leqslant a\uparrow$ there exists ε, $|\varepsilon| < 6\,eps$ such that $a(1+\varepsilon) = b(1+\varepsilon_3)$. The case where $u_y = 0$ or $v_y = 0$ is trivial, since then $t_y = 0$ and the sign of $(u_x - t_x)(-v_y)$ or the sign of $u_y\,(t_x - v_x)$ can be computed exactly which means no perturbation is necessary. Assuming $(u_y \neq 0) \wedge (v_y \neq 0)$ we rewrite $a(1+\varepsilon) = b(1+\varepsilon_3)$ as

$$(u_x - t_x)\cdot(t_y - v_y)\cdot(1+\varepsilon)$$

$$= (u_y - t_y)\cdot(t_x - v_x)\cdot(1+\varepsilon_3)$$

$$\Leftrightarrow (u_x - t_x)\cdot\left(t_y - v_y\cdot\left(1+\varepsilon\cdot\left(1-\frac{t_y}{v_y}\right)\right)\right)$$

$$= \left(u_y\cdot\left(1+\varepsilon_3\cdot\left(1-\frac{t_y}{u_y}\right)\right) - t_y\right)\cdot(t_x - v_x)$$

From $|\varepsilon| < 6\,eps \wedge |\varepsilon_3| < 6\,eps \wedge |1 - t_y/v_y| < 2 \wedge |1 - t_y/u_y| < 2$ we conclude the existence of the 12 *eps* perturbation making the equation exact. ■

6. HOW SHOULD ONE HANDLE DEGENERACIES?

We have seen in the previous discussion that algorithms adhering to the principle of perturbation work well as long as only algorithm-dependent degeneracies are removed. Perturbation is a viable tool for removing these degeneracies. On the other hand, one should not rely on perturbation when dealing with problem inherent degeneracies. Although a post-processing stage which deals with the result delivered by an algorithm based on perturbation is sometimes feasible (e.g. the winding number example) there are cases where post-processing appears to be at least as difficult as solving the problem in the first place (e.g. the line intersection problem). In both situations, however, perturbation will yield an algorithm that is inferior both in speed and simplicity to an algorithm that incorporates the special cases right from the beginning.

Therefore, we propose the following approach.

1. At the beginning there must be a thorough problem analysis and a precise specification of the algorithms input and output, in particular in degenerate cases. Note that often there are many possibilities for specifying the output for degenerate configurations. As an example consider the problem of finding all intersections among line segments $l_1, \ldots, l_n$. We may be interested in the set of intersection points or in the set of all pairs (i, j) such that l_i and l_j intersect. Both specifications are essentially the same as long as there are no degeneracies. The extreme case of all segments having a common intersection point makes the difference between the two output specifications clear: there is one intersection but $n(n-1)$ intersecting pairs.

   Special consideration must also be given to the specification of the input data. In theory geometric algorithms often assume sets as input. In practice this implies eliminating possible duplicates or allowing mutisets as input. The input specification gets even more difficult when more complicated objects such as convex polygons are part of the input. A rigorous specification what constitutes a convex polygon, and

recognizing it, is not a trivial task when all degenerate cases are considered (Schorn and Fisher, 1994).
2. Construct the algorithm with exact (integer) arithmetic and the input–output specification in mind. This means that degenerate cases are taken care of right from the beginning and not as an afterthought. This usually leads to short and efficient programs. Perturbation can conceptually be used to get rid of algorithm-dependent degeneracies.
3. If floating point arithmetic is to be used, the situation becomes much more difficult since only few geometric algorithms are known which behave provably correct when implemented in floating point arithmetic (Mileukovic, 1988, 1989; Fortune, 1992; Schorn, 1994). So far only tedious analysis has yielded usable results.

As a case study adhering to the above guideline we have presented two algorithms for the winding number problem, one with double precision integer arithmetic, the other with floating point arithmetic which can handle all degeneracies adequately.

ACKNOWLEDGEMENTS

I am grateful to Jürg Nievergelt and Klaus Hinrichs for stylistic improvements to this paper.

REFERENCES

Emiris, I. and Canny, J. (1992) An efficient approach to removing geometric degeneracies. In *Proc. Eighth Annual Symp. on Computational Geometry*, pp. 74–82. ACM Press.

Edelsbrunner, H. and Mücke, E. (1990) Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM Trans. Graphics*, **9**, 66–104.

Fortune, S. (1992) Numerical stability of algorithms for 2D delaunay triangulations. In *Proc. Eighth Annual Symp. on Computational Geometry*, pp. 83–92. ACM Press.

Mehlhorn, K. (1993) Degeneracy in geometric computations: curse of blessing. Presentation given at the *ALCOM Workshop *ALGORITHMS: Implementation, Libraries, and Use*τ, IBFI, Dagstuhl.

Milenkovic, V. (1988) *Verifiable Implementations of Geometric Algorithms using Finite Precision Arithmetic*. CMU Report CMU-CS-88-168, Carnegie Mellon, 1988.

Milenkovic, V. (1989) Double precision geometry: a general technique for calculating line and segment intersections using rounded arithmetic. In *Proc. 30th IEEE Symp. on Foundations of Computer Science*, pp. 500–505.

Schorn, P. (1993) An axiomatic approach to robust geometric programs. *J. Symbolic Comput.*, **16**, 155–165.

Schorn, P. and Fisher, F. (1994) How to detect a convex polygon. In Heckbert, P. (ed.), *Graphics Gems IV*. Academic Press, New York.

Salesin, D., Stolfi, J. and Guibas, L. (1989) Epsilon geometry: building robust algorithms from imprecise calculations. In *Proc. Fifth Annual Symp. on Computational Geometry*, pp. 208–217. ACM Press.

Yap, C. (1990) Symbolic treatment of geometric degeneracies. *J. Symbolic Comput.*, **10**, 349–370.